An LSL to PVS Compiler

M.C.A. Devillers

Computing Science Institute/

CSI-R9824 October 1998

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

An LSL to PVS Compiler

M.C.A. (Marco) Devillers

Computing Science Institute
University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
marcod@cs.kun.nl

October 20, 1998

ABSTRACT

A compiler which translates axiomatic specifications written in the Larch Shared Language (LSL) to theories of the Prototype Verification System (PVS) is described by means of examples. Besides giving an equivalent axiomatic specification in PVS, the compiler enables one to check that an LSL specification has a model in the higher-order logic of PVS.

1994 Extended Computing Reviews Classification System: D.2.4 [Software Engineering]: Program Verification - Assertion checkers, correctness proofs, F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying and Reasoning, F.4.3 [Formal Languages]: Mathematical Logic and Formal Languages - Algebraic Language Theory, Operations on Languages

1991 Mathematics Subject Classification: 03B10 Classical First-Order Logic, 03B15 Higher-Order Logic and Type Theory, 03B15 Foundations and Axiomatics of Classical Theories, 03B35 Mechanization of proofs and logical operations,

Keywords and Phrases: Algebraic Specifications, Tools, Theorem Provers

Table of Contents

1. Introduction

The Larch Shared Language (LSL)[GH93] is an axiomatic specification language based on a multi-sorted first-order logic with equality. Theories axiomatically specified in LSL, by means of traits, are similar to conventional algebraic specifications; in these traits, sorts and operators are introduced together with assertions (properties which are known to hold) and consequences (properties which are thought to be derivable). Normally, traits are translated to input for the Larch Prover (LP), a first order theorem prover, where one can prove the intended consequences from the asserted properties given in the trait. The Prototype Verification System (PVS)[ORSH95] is a theorem prover based on classic higher-order logic. In this paper, we describe a compiler which translates LSL traits to theories of the PVS language.

There is a well-known problem with axiomatic specifications: inconsistent specifications are easily given in an axiomatic specification language, this is also discussed by Jim Horning in [Hor96]. The compiler helps to address this problem by translating LSL traits to two PVS theories: besides an axiomatization of the trait in which stated consequences are to be proven, a second theory is generated in which it is to be shown that the axiomatization can be modeled by a conservative extension.

There are a number of ways in which the compiler can be used. The most interesting is that users of LP may use the compiler to show that an axiomatic specification has a model within the logic of PVS. Also, users of PVS can use the compiler to work with LSL theories; or, it is possible to use the compiler to have properties proven by either LP or PVS.

In this paper, no formal claims are made about semantical equivalence of LSL traits and PVS theories generated; an important question to be addressed when combining two proof tools. However, since LSL has straightforward set-theoretic semantics, a simple translation to PVS was easily found which

we believe preserves semantical equivalence.

The outline of the paper is as follows. In section two, a number of examples show how LSL traits are translated to PVS. In the third section, we explain work done and future work on the LSL compiler. We end with the conclusions in the last section, section four.

2. Translating LSL to PVS

In this section we informally describe LSL and the translation to PVS.

2.1 Traits

First, let us consider a simple example, the specification of a group.

A group is an algebra $< S, o, 1 >$ in which

1. o is a binary, associative operator,
2. o has the identity 1 in S,
3. every element b of S has an inverse, denoted as b^{-}.

As stated before, LSL supports a multi-sorted first-order logic with equality. Sorts are non-empty sets of objects. Below we show the LSL specification of a group. LSL's basic unit of specification is a trait; the name of the trait in the example is `Group`. In the introduces clause, the operators `o`, `unit`, and `inv` are declared; the sort `S` is implicitly declared by its occurrence. The `asserts` clause defines their properties conform the definition given above; in the `implies` clause, purported consequences of these properties are given.

```
Group : trait
  introduces
    o:    S, S -> S,
    unit:      -> S,
    inv:  S    -> S
  asserts with a,b,c : S .
    o(o(a, b), c) == o(a, o(b, c));
    o(unit, a) == a;
    o(a, unit) ==  a;
    o(inv(a), a) == unit;
    o(a, inv(a)) == unit
  implies with a,b,c,d,x : S .
    b == inv(inv(b));
    o(b,d) = o(c,d) == b = c;
    o(d,b) = o(d,c) == b = c;
    o(b,x) = c == x = o(inv(b),c);
    o(x,b) = c == x = o(c, inv(b));
    b ~= c == o(d,b) ~= o(d,c);
    b ~= c == o(b,d) ~= o(c,d);
    \E x: S . o(b,x) = c;
    \E x: S . o(x,b) = c
```

First, we show the axiomatic theory generated in PVS below. As stated in the trait, `S` is a nonempty set (as specified by the PVS keyword `TYPE+`, and the operators `o`, `unit`, `inv` are introduced as uninterpreted constants. Meaning is given to the constants by a number of `AXIOM`s, the translation of the assertions of the trait; and intended consequences are translated to `LEMMA`s.

```
Group: THEORY
  BEGIN

    S : TYPE+;
    o : [[S, S]->S];
    unit : S;
    inv : [[S]->S]

    assertion0 : AXIOM FORALL (a : S, b : S, c : S) : (o(o(a,b),c) = o(a,o(b,c)));
    assertion1 : AXIOM FORALL (a : S) : (o(unit,a) = a);
    assertion2 : AXIOM FORALL (a : S) : (o(a,unit) = a);
    assertion3 : AXIOM FORALL (a : S) : (o(inv(a),a) = unit);
    assertion4 : AXIOM FORALL (a : S) : (o(a,inv(a)) = unit);

    consequence0 : LEMMA FORALL (b : S) : (b = inv(inv(b)));
    consequence1 : LEMMA FORALL (b : S, d : S, c : S) : ((o(b,d) = o(c,d)) = (b = c));
    consequence2 : LEMMA FORALL (d : S, b : S, c : S) : ((o(d,b) = o(d,c)) = (b = c));
    consequence3 : LEMMA FORALL (b : S, x : S, c : S) : ((o(b,x) = c) = (x = o(inv(b),c)));
    consequence4 : LEMMA FORALL (x : S, b : S, c : S) : ((o(x,b) = c) = (x = o(c,inv(b))));
    consequence5 : LEMMA FORALL (b : S, c : S, d : S) : ((b /= c) = (o(d,b) /= o(d,c)));
    consequence6 : LEMMA FORALL (b : S, c : S, d : S) : ((b /= c) = (o(b,d) /= o(c,d)));
    consequence7 : LEMMA FORALL (b : S, c : S) : EXISTS (x : S) : (o(b,x) = c);
    consequence8 : LEMMA FORALL (b : S, c : S) : EXISTS (x : S) : (o(x,b) = c);

  END Group
```

We show the proof of "for all b in S we have $b \quad = (b^-)^-$" (`consequence0`) mathematically together with the corresponding PVS proof script.

Take an arbritrary b in S `(SKOLEM 1 "b")`. Then

$(b^-)^-$	=	$<$ 1 is the left identity $>$
		`(LEMMA "assertion1" ("a" "inv(inv(b))")) (REPLACE -1 1 RL :HIDE? T)`
$1o(b^-)^-$	=	$<$ definition of right inverse $>$
		`(LEMMA "assertion4" ("a" "b")) (REPLACE -1 1 RL :HIDE? T)`
$(bob^-)o(b^-)^-$	=	$<$ associativity o $>$
		`(USE "assertion0") (REPLACE -1 1 :HIDE? T)`
$bo(b^-o(b^-)^-)$	=	$<$ definition of right inverse $>$
		`(USE "assertion4") (REPLACE -1 1 :HIDE? T)`
$bo1$	=	$<$ 1 is the right identity $>$
		`(USE "assertion2") (ASSERT)`
b		

Below, we show a generated "model" theory; the user is expected to provide (type in) instantiations for the uninterpreted constants such that the lemmas generated hold. Of course, we expect that instantiations given are taken (or constructed) out of the logic of PVS in a conservative manner: i.e., we expect them to be well-defined. In this case, we show that $<\mathbb{Z}, +, 0>$ is a group by providing the necessary instantiations. In the example, all proofs of the lemmas are trivially done by PVS with the proof strategy `(GRIND)`.

```
GroupModel: THEORY
  BEGIN

    S : TYPE+         = int;
    o : [[S, S]->S]   = +;
```

```
    unit : S          = 0;
    inv : [[S]->S]    = -;

    assertion0 : LEMMA FORALL (a : S, b : S, c : S) : (o(o(a,b),c) = o(a,o(b,c)));
    assertion1 : LEMMA FORALL (a : S) : (o(unit,a) = a);
    assertion2 : LEMMA FORALL (a : S) : (o(a,unit) = a);
    assertion3 : LEMMA FORALL (a : S) : (o(inv(a),a) = unit);
    assertion4 : LEMMA FORALL (a : S) : (o(a,inv(a)) = unit);

  END GroupModel
```

2.2 Inductive Sorts

The previous example was very basic. LSL allows one to specify more complex sorts, such as sorts which are inductively generated by a set of operators. Below we give an example, a trait specifying a sort of unary numbers `U` by means of two generators: an initial constructor `0` and a successor operator `Succ`.

```
UnaryNumbers : trait
  introduces
    0:      -> U,
    Succ: U -> U
  asserts
    sort U generated by 0: -> U, Succ: U -> U
```

The `generated by` assertion states that every object in `U` can be generated by a finite number of (legal) applications of the two generators `0` and `Succ`.

Below the PVS axiomatization of the above trait is given. Two axioms are generated: `U_cover` states that every object in `U` can be constructed from one of the two generators, the next axiom `U_induction` states the induction proof principle over the generators. Note that the first axiom follows from the latter; however, we prefer to give both for convenience.

```
UnaryNumbers: THEORY
  BEGIN
    U : TYPE+;
    0 : U;
    Succ : [[U]->U]


    U_cover : AXIOM FORALL (u0 : U) : ((0 = u0) OR EXISTS (u1 : U) : (u0 = Succ(u1)));

    U_induction : AXIOM FORALL (P : [[U]->bool]) :
       ((P(0) AND FORALL (u0 : U) : (P(u0) => P(Succ(u0)))) => FORALL (u0 : U) : P(u0));


  END UnaryNumbers
```

However, the above specification is too weak as it does not state that all unary numbers generated are different. This can be specified by adding the keyword `freely` to the specification. This states that every object in the sort is uniquely described by a finite number of applications of the generators.

```
    sort U generated freely by 0: -> U, Succ: U -> U
```

If the `freely` keyword is added to the `generated by` assertion, two extra axioms are generated.

```
U_disjoint : AXIOM
  FORALL (u0 : U) : NOT((0 = Succ(u0)));

U_injective : AXIOM
  FORALL (u0 : U, u1 : U) : ((Succ(u0) = Succ(u1)) => (u0 = u1));
```

2.3 Observers on sorts

In LSL, one can state that a set of operators is a set of observers for a sort; that is, each two objects in a sort which give similar results under the observers are equal. Below, an example is given. The example states that two states of a lamp are the same if you cannot observe a difference by looking at it.

In the trait below, the `partitioned by` assertion states that `look` is an observer over the sort`States`.

```
Lamp : trait
  introduces
    on,off:  State  -> State,
    look:    State  -> bool
  asserts  with s : State.
    sort State partitioned by look : State -> bool;
    look(on(s));
    ~look(off(s))
```

The assertion above is translated to the following axiom.

```
State_partition : AXIOM FORALL (s_var0 : State, s_var1 : State) :
   ((look(s_var0) = look(s_var1)) => (s_var0 = s_var1));
```

2.4 Reusing specifications

LSL allows traits to be reused in other traits in two manners: by including them, in this case the sorts and operators defined in the included traits are added to the including trait, or by assuming traits, then the axioms of the assumed trait are added as assumptions to the assuming trait.

Whenever a trait is included or assumed, sorts and operators in that trait may be renamed. An included or assumed sort coincides with another sort if, possibly after renaming, they have the same name; an included or assumed operator coincides with another operator if, possibly after renaming, they have the same name and the same type. Coinciding sorts, or operators, are understood to denote the same sort, or operator.

Including traits In the example below, the inclusion (and renaming) mechanism of LSL is demonstrated. A sort, or operator, may always be renamed by means of the `for` construct. However, sometimes it is more convenient to provide a trait with a number of parameters of often renamed sorts, or operators. This is shown in the second trait named `transitive`.

```
irreflexive : trait
  introduces
    less:T, T->bool
  asserts with x:T .
    ~less(x, x).
```

```
transitive (S, less:S,S->bool) : trait
  introduces
    less:S, S->bool
  asserts with x, y, z:S .
    less(x, y)/\less(y, z)=>less(x, z).

partialorder(R, less:R,R->bool) : trait
  includes
    irreflexive(R for T, le:R,R -> bool for less:T,T ->bool),
    transitive(R, le:R,R->bool).
```

In the resulting theory for the the `partialorder` trait, one sort `R` and one operator `le` are axiomatically described.

```
partialorder: THEORY
  BEGIN
    R : TYPE+;
    le : [[R, R]->bool]

    assertion0 : AXIOM FORALL (x : R) : NOT(le(x,x));
    assertion1 : AXIOM FORALL (x : R, y : R, z : R) : ((le(x,y) AND le(y,z)) => le(x,z));

   END partialorder
```

We would like to note that if a trait is included or assumed, the consequences of that trait become axioms in the including or assuming trait since they are supposed to be derivable.

Assuming traits Since one can include traits in other traits, it is often convenient to describe that a trait may only be included in a certain context, i.e. when a number of assumptions hold. For instance, in the example below, trait `reflexiveclosure` states that a reflexive closure operator `lesseq` only exists whenever $<$ `T, less` $>$ is a partial order.

```
reflexiveclosure (T, less:T,T->bool) : trait
  assumes partialorder(T, less:T,T->bool)
  introduces
    lesseq:T, T->bool
  asserts with x,y:T .
    lesseq(x, y) == IF x=y THEN true ELSE less(x,y) FI
  implies
    trait transitive (T, lesseq:T,T->bool).
```

In the trait `reflexiveclosure`, the axioms of the assumed trait are assumed to hold. However, when trait `natreflexive`, shown below, includes this trait, the axioms of trait `partialorder` become proof obligations in the including trait.

```
natreflexive : trait
  includes reflexiveclosure(nat for T)
  asserts with x,y : nat .
     less(x,y) == x < y
  implies with x,y : nat .
     lesseq(x,y) == x <= y.
```

The translation of trait `natreflexive` now becomes non-trivial to read; it is show below. First, the assertions of the trait are listed as axioms. In this case, the only assertion is a definition for the `less` operator. After the assertions, the to be proven assumptions of the included trait `reflexiveclosure` are listed as lemmas. Then, the assertions and the consequences of the included trait are listed as axioms, since they are known to hold. Lastly, the only intended consequence stated in the trait `natreflexive` is listed as a lemma.

```
natreflexive: THEORY
  BEGIN
    less : [[nat, nat]->bool];
    lesseq : [[nat, nat]->bool]

    assertion0 : AXIOM
       FORALL (x : nat, y : nat) : (less(x,y) = (x < y));

    assumption0 : LEMMA
       FORALL (x : nat) : NOT(less(x,x));
    assumption1 : LEMMA
       FORALL (x : nat, y : nat, z : nat) : ((less(x,y) AND less(y,z)) => less(x,z));

    assertion1 : AXIOM
       FORALL (x : nat, y : nat) : (lesseq(x,y) = IF (x = y) THEN true ELSE less(x,y) ENDIF);

    consequence0 : AXIOM
       FORALL (x : nat, y : nat, z : nat) : ((lesseq(x,y) AND lesseq(y,z)) => lesseq(x,z));
    consequence1 : LEMMA
       FORALL (x : nat, y : nat) : (lesseq(x,y) = (x <= y));

  END natreflexive
```

Note that in the previous trait the sort `nat` is not introduced since it is, as is the sort `bool`, a predefined sort of LSL.

Discharging assumptions A drawback of including or assuming traits is that assertions or consequences are easily included multiple times through several "paths". However, when possible, superfluous (syntactically equivalent) properties are discharged by the compiler.

2.5 Shorthands

Shorthands abbreviate often used specifications. LSL has shorthands for enumerations, tuples and unions. On the following pages, examples of these shorthands are given together with their intended meaning.

Enumerations In the example below, the `enumeration` shorthand is used to define the sort `YRB` as an enumeration of `yellow`, `red`, and `blue`.

```
Kandinski : trait
  sort YRB enumeration of yellow, red, blue.
```

The above trait is understood to be equivalent to the trait below.

```
Kandinski :trait
```

```
  introduces
    yellow, red, blue:     -> YRB,
    succ:  YRB -> YRB
  asserts
    sort YRB generated freely by yellow, red, blue;
    succ(yellow) = red;
    succ(red) = blue
```

Tuples In the example below, the `tuple` shorthand is used to define the sort `Edge` as an pair of `Node`s named `in` and `out`.

```
Graph : trait
  sort Edge tuple of in: Node, out: Node.
```

The above trait is understood to be equivalent to the trait below.

```
Graph: trait
 introduces
   T:        Node, Node -> Edge,
   in: Edge -> Node,
   out:Edge -> Node,
   set_in:   Edge, Node -> Edge,
   set_out:  Edge, Node -> Edge
 asserts with n0,n1,n2 : Node .
   sort Edge generated freely by T: Node, Node->Edge;
   sort Edge partitioned by in, out: Edge -> Node;
   in(T(n0,n1)) = n0;
   out(T(n0,n1)) = n1;
   set_in(T(n0,n1), n2) = T(n2,n1);
   set_out(T(n0,n1), n2) = T(n0,n2).
```

Unions Below, the `union` shorthand defines the sort `Possession` as a distinct product of `yours` or `mine` elements in the sort `T`.

```
Possession(T) : trait
  sort Possession[T] union of yours: T, mine : T.
```

The above trait is understood to be equivalent to the trait below.

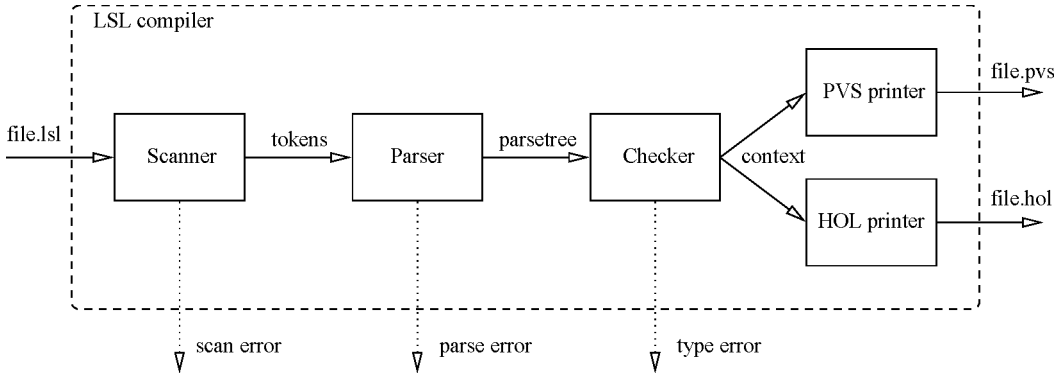
```
Possession: trait
  sort tag[Possession[T]] enumeration of yours, mine
  introduces
    tag:       Possession[T] -> tag[Possession[T]],
    yours:     T             -> Possession[T],
    mine:      T             -> Possession[T],
    get_yours: Possession[T] -> T,
    get_mine:  Possession[T] -> T
  asserts with t0 : T.
    sort Possession[T] generated by
       yours, mine: T -> Possession[T];
    sort Possession[T] partitioned by
       tag: Possession[T] -> tag[Possession[T]],
       get_yours, get_mine:  Possession[T] -> T;
```

```
tag(yours(t0)) = yours;
tag(mine(t0)) = mine;
get_yours(yours(t0)) = t0;
get_mine(mine(t0)) = t0.
```

3. Related and Future Work

The compiler described is intended to become part of the IOA toolset [GL98]. The toolset is developed around the IOA language, a language for describing distributed systems as input/output automata according to the model of Lynch and Tuttle [LT89] [Lyn96]. Tools under development are a simulator, a compiler to the Larch theorem prover[GH93] and a compiler to the model checker SPIN[Hol91]. My interest is to compile the IOA language to PVS and, if possible, to Isabelle/HOL[Pau94]. Since LSL is part of the IOA language, the LSL to PVS compiler has been developed as a first step in this project. We have chosen to implement a new compiler to be able to support possible future higher-order logic extensions.

Below a picture describing the structure of the compiler is shown. The compiler takes a number of LSL specifications as its input and translates it in a number of steps to a PVS theory files. The boxes describe several components of the compiler, and the arrows the data structures passed between them. Also, it is shown which components can produce errors. The component which translates to an Isabelle/HOL specification file has not been implemented yet.



Below, we discuss different parts of the tool and related work.

- The *scanner* takes an input file and translates it into a sequence of tokens. It was implemented with the `javacc` compiler generator.
- The *parser* recognizes a grammar and transforms it into a parse tree. The grammar recognized is an LL(2) subset of the original LSL language to ease the translation. This grammar is developed with use of the Grammar Work Bench[NKDvZ92]; a tool which we also use to generate terms in the LSL language to test the parser implemented. Since the language recognized is LL(2), it is implemented in a straightforward manner as a recursive descent parser.
- The *checker* translates the parse tree into a context. The checker checks semantic consistency of the input and performs the actual translation. The context build describes translated input as a collection of terms in an intermediate higher-order logic language. The intermediate language is used to be able to easily translate to different (higher-order logic) proof tools. This intermediate language and the translation is described in a paper available at `http://www.cs.kun.nl/~marcod/`. In this paper, the specification of the checker is given as a

transducer denoted as an attribute grammar where attributes range over terms in this intermediate language.

- The *PVS printer* or the *HOL printer* translate the intermediate representation into either PVS or Isabelle/HOL specification files.

The LSL compiler has been developed in Java[AG] in order to be able to easily link this compiler to other tools in the toolset. The following line describes how to start the tool:

```
java lsl [-/+check][-/+tokens][-/+unparse][-/+context][-/+pvs][-/+hol] fn.lsl
```

The `check` switch may be used to semantically check an input file. The other switches produce the output of one of the components refered to. For instance, the `context` switch will produce a printed representation of the context, or the `pvs` switch will produce PVS output. The compiler is lazy in the sense that if only the `unparse` switch is set, for instance, it will only print the parse tree but will not perform semantical analysis.

We see two obvious extensions to the LSL tool. One is the addition of labeling of the axioms and consequences denoted in the original LSL specification to facilitate proof development in the theorem provers compiled to. For instance, at the moment, it reads awkward that a theorem named `consequence4` holds by application of `assertion2` and `consequence0`. The second extension would be to automatically generate proof strategies. For instance, proof strategies for induction principles generated may automate part of proving efforts.

However, future work will be mainly in the IOA toolset context. The attribute grammar for LSL is to be extended to an attribute grammar which describes a transducer for the whole IOA language, and the LSL compiler is to be extended to an IOA compiler.

4. Conclusions

We have presented a compiler which translates axiomatic specifications given in the Larch Shared Language to two theories for PVS: one theory in which it is possible to reason about the axiomatic specification given, and a second theory in which it can be shown that the axiomatic specification has a model. In this manner, a well-known problem with LSL, it is very easy to write down inconsistent axiomatic specifications, can be addressed. No formal claims are made about semantic equivalence of input and output theories; however, it is argued that, since input and output theories have the same set-theoretic underpinnings, we believe that they are semantically equivalent.

The LSL compiler has been developed as part of a toolset for the IOA specification language of which the LSL language forms a subset. There are a number of interesting features which can be implemented in the compiler; either for LSL users (conservative extensions, subsorting) or for PVS users (generation of proof strategies). However, the compiler is first to be extended to recognize and translate the total IOA language.

Note: A beta-release of the compiler can be tried at `http://www.cs.kun.nl/~marcod/lsl.html`

References

[AG] Ken Arnold and James Gossling. *The Java Programming Language.* Addision Wesley.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GL98] S.J. Garland and N.A. Lynch. The ioa language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Massachusetts Institute of Technology, August 1998.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hor96] J.J. Horning. the larch shared language: Some open problems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[LT89] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[NKDvZ92] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The grammar workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and H.J. op den Akker, editors, *Linguistic Engineering: Tools and Products*, volume 92-99 of *Memoranda Informatica*, pages 103–115. Springer-Verlag, April 1992.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.